Creating and optimizing a custom effect for the Nokia Imaging SDK

This article explains how to create a custom effect for the Nokia Imaging SDK, as well as optimize the effect's performance by moving it to native code. The article provides guidance on how to transform an existing algorithm to take advantage of SIMD instructions like ARM NEON. We'll also be building a small test bench application to compare the optimized versions with the reference implementation.

29 Dec 2013

Note: This is an entry in the Nokia Imaging and Big UI Wiki Competition 2013Q4.

Introduction

As the starting point of our discussion, we will be using a variant of the DoubleEffect that Nokia provides. The variant we'll use is called the MultiplyEffect. It multiplies each color component with a multiplication factor provided to the Effect as a parameter. The alpha channel will not be altered by the effect. The reason for using such a simple effect is to keep the resulting code simple enough to make it easy to understand what we are doing in each step.

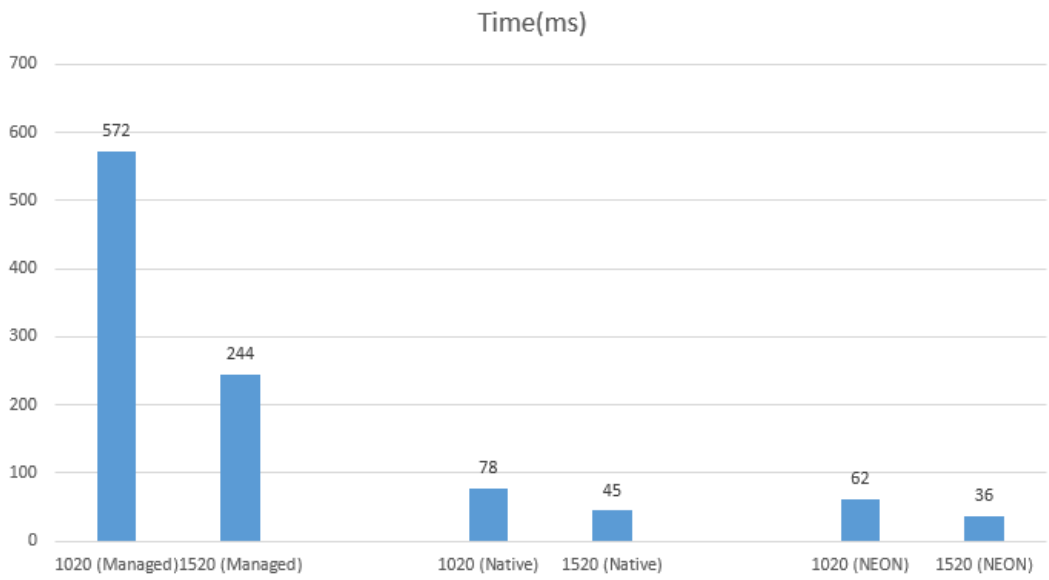
Our first implementation will be done in C# to take advantage of the `Nokia.Graphics.Imaging.CustomEffectBase` base class provided by the SDK. It will serve as our reference implementation. We will use it to check whether the results provided by our optimized filters are the same.

Next, we will re-implement the `MultiplyEffect` as a WinPRT component written in C++/Cx. Since we won't be able to use the `CustomEffectBase`, we will have to implement the `Nokia::Graphics::Imaging::ICustomEffect` interface.

Once we have these two implementations of the `MultiplyEffect`, we will compare them. We will talk about what kind of information each implementation outputs, and why that information is helpful when trying to optimize an imaging effect. We will not look into the implementation of the comparer, since it's quite easy to read the code provided as part of the sample project.

This part of the article will present techniques for transforming your code to make it easier to optimize using Vector Processing, and give examples of how to optimize using ARM NEON intrinsics.

The following diagram shows the performance impacts of all of these steps. Keep in mind though that these are benchmarking numbers applicable to the `MultiplyEffect` on the tested devices. The speedup may be bigger or smaller, depending on the effect you implement. For some of the effects, the performance will depend on the contents of the processed image, in addition to its size.



The rest of the article will assume that you have at least a basic understanding of both C# and C++ code. I will provide pointers to useful resources that outline special behaviors or functionality required to understand the code. As the article aims to outline possible steps to get better performance instead of trying to just showcase optimized code, the resulting code still has potential for more optimization.

If you have not yet worked with the Imaging SDK, you can find instructions on how to install it in your project in Nokia's documentation: Download and add the libraries to the project. If you have multiple projects, you will be asked to select the ones

(C) Copyright Nokia 2014. All rights reserved.

that will use the Imaging SDK. Sometimes compiling against the SDK will fail immediately after download. If that happens, a restart of Visual Studio will fix the problem.

Note: The sample solution contains ARM NEON intrinsics. The resulting code can only run on ARM CPUs; it will not run in the emulator. Therefore the affected code has been placed in a separate project called CustomEffectNativeNeonOptimized. This project can easily be removed from the solution to test the rest of the sample without having to use an actual device.

Reference Implementation in C#

You can easily create a custom effect in C# by making your effect class inherit from `Nokia.Graphics.Imaging.CustomEffectBase`. You only need to implement the `OnProcess` method to create a new effect. Depending on the settings of the imaging pipeline, your code should not always alter the entire image, so the parameters of `OnProcess` give you direct access to only the pixel data that you need to alter.

We've defined a helper method called `MultiplyBounded`. `MultiplyBounded` does the actual calculation for each color component of a pixel, while making sure that the resulting value is no larger than 255. We only have 8 bits to represent each color component. We ask the end user of our `MultiplyEffect` to set a multiplication factor, which will be provided to the effect's constructor.

```
using System;
using Windows.UI;
using Nokia.Graphics.Imaging;

namespace CustomEffectManaged
{
    public class MultiplyEffect : CustomEffectBase
    {
        private byte multiplier;

        public MultiplyEffect(IImageProvider source, byte multiplier) : base(source)
        {
            this.multiplier = multiplier;
        }

        protected override void OnProcess(PixelRegion sourcePixelRegion, PixelRegion
targetPixelRegion)
        {
            sourcePixelRegion.ForEachRow((index, width, pos) =>
            {
                for (int x = 0; x < width; ++x, ++index)
                {
                    Color c = ToColor(sourcePixelRegion.ImagePixels[index]);
                    c.R = MultiplyBounded(c.R);
                    c.G = MultiplyBounded(c.G);
                    c.B = MultiplyBounded(c.B);
                    targetPixelRegion.ImagePixels[index] = FromColor(c);
                }
            });
        }

        private byte MultiplyBounded(byte value)
        {
            return (byte)Math.Min(255, value * multiplier);
        }
    }
}
```


(C) Copyright Nokia 2014. All rights reserved.

The code of the C# implementation of our `MultiplyEffect` can be found inside the CustomEffectManaged project included with the sample solution.

Basic Implementation in C++

Creating a custom effect in C++ is a little more complicated than in C#. You can't directly inherit from a base class and just add the code required to transform the input to the output. Instead, you have to make your WinRT component implement the interface `Nokia::Graphics::Imaging::ICustomEffect`. So that you don't have to implement everything yourself, the resulting component is not an effect in it's own right. You have to wrap it inside a `Nokia.Graphics.Imaging.DelegatingEffect` to execute it.

The following code is the basic template you need to start building a custom filter in native code. Aside from the class/namespace names of the implementation you can copy/paste it to start work on your own custom effect written in native code.

Header file:

```
// MultiplyEffect.h
#pragma once

namespace CustomEffectNative
{
    public ref class MultiplyEffect sealed : public
Nokia::Graphics::Imaging::ICustomEffect
    {
    public:
        MultiplyEffect();
        virtual Windows::Foundation::IAsyncAction^ LoadAsync();
        virtual void Process(Windows::Foundation::Rect rect);
        virtual Windows::Storage::Streams::IBuffer^
ProvideSourceBuffer(Windows::Foundation::Size imageSize);
        virtual Windows::Storage::Streams::IBuffer^
ProvideTargetBuffer(Windows::Foundation::Size imageSize);
    private:
       unsigned int imageWidth;
       Windows::Storage::Streams::Buffer^ sourceBuffer;
       Windows::Storage::Streams::Buffer^ targetBuffer;
    };
}
```

Implementation:

```
// MultiplyEffect.cpp
#include "pch.h"
#include <wrl.h>
#include <robuffer.h>
#include "MultiplyEffect.h"
#include <ppltasks.h>
#include "Helpers.h"

using namespace CustomEffectNative;
using namespace Platform;
using namespace concurrency;
using namespace Nokia::Graphics::Imaging;
using namespace Microsoft::WRL;

MultiplyEffect::MultiplyEffect()
{
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
Windows::Foundation::IAsyncAction^ MultiplyEffect::LoadAsync()
{
 return create_async([this]
 {
  //add your initialization logic here
 });
}

void MultiplyEffect::Process(Windows::Foundation::Rect rect)
{
 //code implementing the actual effect will go here
}

Windows::Storage::Streams::IBuffer^
MultiplyEffect::ProvideSourceBuffer(Windows::Foundation::Size imageSize)
{
 unsigned int size = (unsigned int)(4 * imageSize.Height * imageSize.Width);
 sourceBuffer = ref new Windows::Storage::Streams::Buffer(size);
 sourceBuffer->Length = size;
 imageWidth = (unsigned int)imageSize.Width;
 return sourceBuffer;
}

Windows::Storage::Streams::IBuffer^
MultiplyEffect::ProvideTargetBuffer(Windows::Foundation::Size imageSize)
{
 unsigned int size = (unsigned int)(4 * imageSize.Height * imageSize.Width);
 targetBuffer = ref new Windows::Storage::Streams::Buffer(size);
 targetBuffer->Length = size;
 return targetBuffer;
}
```

Note: The above code frequently shows the hat-symbol '^' at the end of a variable name. This signifies that the value is a reference-counted WinRT component. Such components are declared as "public ref class <classname> sealed" and instantiated using "ref new" instead of "new". A more detailed description of how this works is outside the scope of this article. For now all you need to know is that when you declare a class as a WinRT component, the Runtime will make it available for use from all WinRT languages. This includes the Managed languages C# and Visual Basic. A good introduction of these extensions to C++, called C++/Cx, can be found in the article A Tour of C++/CX Note that the samples in this article use Windows 8 rather than Windows Phone 8.

As you can see we have to do quite a bit more work than in C#. First, we have to provide Buffers to the SDK that are used to store the input and output data. You might have noticed that as part of the class declaration we added a field called `imageWidth`. While the calculation of our `MultiplyEffect` does not need to have knowledge of any pixel except the current pixel, many other effects do need to know about the entire image. To be able to calculate the location of other pixels, such as the pixel above the current one, the width needs to be known. We will also need the width so the method only works with the pixels inside the buffers that we want to alter. (In C# we used `sourcePixelRegion` for this.)

Another method that we have to implement as part of the interface, but won't be needing in this example, is `LoadAsync`. `LoadAsync` is used to preload and precompute assets that don't depend on the input image. For example, you might use this to blend the image you get from the rendering pipeline with another image. The other image would be passed to your effect as a parameter. You could load it into memory when `LoadAsync` is called, so it would be immediately available when you need it for the actual rendering.

When you inspect the buffers that we are creating, you might notice that their interface does not provide any access to the encapsulated raw data. To gain access to this, you will have to perform some recasting of COM objects. Fortunately Microsoft provides the code for this in the article Obtaining pointers to data buffers (C++/CX). I have placed the described function in a

 (C) Copyright Nokia 2014. All rights reserved.

separate helper namespace, since it's used in several different versions of the `MultiplyEffect`.

```
// Helpers.cpp
#include "pch.h"
#include <wrl.h>
#include <robuffer.h>
#include "Helpers.h"
#include <ppltasks.h>

using namespace Microsoft::WRL;

//method definition from http://msdn.microsoft.com/en-
us/library/Windows/Apps/dn182761.aspx
byte*
CustomEffectNative::Helpers::GetPointerToPixelData(Windows::Storage::Streams::IBuffer^
pixelBuffer, unsigned int *length)
{
    if (length != nullptr)
    {
        *length = pixelBuffer->Length;
    }
    // Query the IBufferByteAccess interface.
    ComPtr<Windows::Storage::Streams::IBufferByteAccess> bufferByteAccess;
    reinterpret_cast<IInspectable*>(
pixelBuffer)->QueryInterface(IID_PPV_ARGS(&bufferByteAccess));

    // Retrieve the buffer data.
    byte* pixels = nullptr;
    bufferByteAccess->Buffer(&pixels);
    return pixels;
}
```

Now that all the support code is in place, we can concentrate on how to implement the actual effect. We'll use the same implementation of the methods that we used in the C++ section for all the examples in the rest of this article.

```
void MultiplyEffect::Process(Windows::Foundation::Rect rect)
{
 unsigned int sourceLength, targetLength;
 byte* sourcePixels = CustomEffectNative::Helpers::GetPointerToPixelData(sourceBuffer,
&sourceLength);
 byte* targetPixels = CustomEffectNative::Helpers::GetPointerToPixelData(targetBuffer,
&targetLength);

 unsigned int minX = (unsigned int)rect.X * 4;
 unsigned int minY = (unsigned int)rect.Y;
 unsigned int maxX = minX + (unsigned int)rect.Width * 4;
 unsigned int maxY = minY + (unsigned int)rect.Height;

 for(unsigned int y = minY; y < maxY; y++)
 {
  unsigned int xOffset = y * imageWidth * 4;
  for(unsigned int x = minX; x < maxX; x += 4)
  {
   //Imaging SDK uses Blue, Green, Red, Alpha Image Format with 8 bits/channel
   byte b = sourcePixels[xOffset + x];
```


(C) Copyright Nokia 2014. All rights reserved.

```
    byte g = sourcePixels[xOffset + x + 1];
    byte r = sourcePixels[xOffset + x + 2];
    byte a = sourcePixels[xOffset + x + 3];

    b = MultiplyBounded(b);
    g = MultiplyBounded(g);
    r = MultiplyBounded(r);

    targetPixels[xOffset + x] = b;
    targetPixels[xOffset + x + 1] = g;
    targetPixels[xOffset + x + 2] = r;
    targetPixels[xOffset + x + 3] = a;
   }
  }
 }

 byte MultiplyEffect::MultiplyBounded(byte value)
 {
  return (byte)min(255, ((int)value) * multiplier);
 }
```

The implementation of the effect is still pretty straightforward. First we use the `GetPointerToPixelData` helper method to extract the actual data arrays from the buffers. Next we calculate the bounds of the area we want to alter, so we can use the bounds in the loops that process the image data. Each pixel consists of 4 bytes of data (Blue, Green, Red and Alpha). Since the image width is given in pixels, we have to multiply all width-related values by 4. We need to move through the image along its x-Axis using +4 increments for the same reason.

We extract the data for all color channels to separate variables, apply the calculations to each of them using `MultiplyBounded`, and finally, store the data in the output buffer. This is pretty much the same as in the C# version, except that we can't use the `ToColor`/`FromColor` helper methods. Therefore I opted to do it without using Color structs as well.

Note: Regular C++ programmers might have noticed that the sample code makes use of the datatype byte which does not exist in C++. It is a custom typedef of the Windows Runtime and can safely be exchanged for unsigned char in your own code.

The full code of the C++ implementation of our `MultiplyEffect` can be found inside the CustomEffectNative project included with the sample solution.

Comparing Effects

The user interface of our sample application is designed to help you test and compare different versions of your own custom effect. The first page of the application contains a button that allows the user to select an image from the media library. Once the

(C) Copyright Nokia 2014. All rights reserved.

image is selected, two effects will be applied to the selected image: our reference implementation, as well as our optimization implementation. The results of the reference filter will be shown above the button, and the results of the comparison filter will be shown below. That way we can see potential issues at a glance.

Performance

The second page of the application is likely to be of the most interest when dealing with effect optimization. It shows the time it took to execute the reference effect and the comparison effect in milliseconds. It also shows some information on the size of the image. It's important to note that those values will vary, not only from device to device, but also, to a certain degree, between consecutive runs on the same device. Therefore, to really compare your changes, you will have to run the test a few times. The following table shows the minimum and maximum time it took to run the two effects we implemented on a Lumia 1020, as well as the minimum time it took them to execute on a Lumia 1520. The image we used has a size of 1188 by 1188 pixels and is included with the sample project.

Effect	Lumia 1020 max	Lumia 1020 min	Lumia 1520 min
Managed (C#)	745 ms	708 ms	346 ms
Native (C++)	370 ms	358 ms	238 ms

In the sample project, the `BitmapComparer` class does the performance measurement. `BitmapComparer` builds a minimal imaging pipeline for the image and effects that are provided to its `Compare` method. The time measured is the time it takes to complete the `RenderAsync` call; rendering the provided `Bitmap` using the provided effects to another `Bitmap` of matching dimensions. This is the closest we can get to the time it takes for the actual effect to run without putting the benchmarking logic directly inside the effect.

It is quite easy to use `BitmapComparer` for your own effects. You just have to instantiate your effects by wrapping native effects inside a `DelegatingEffect`), and call the `Compare` method. Since the custom effects used in `BitmapComparer` are WinRT components, you can simply instantiate them as you would any managed class. The Windows Phone Runtime will take care of all marshaling, etc. behind the scenes. The best place to instantiate your effects is in the `DoComparison` method of the `ComparisonPage` included in the sample project. All effects developed as part of this article are also instantiated there, to make it easier to test everything out.

```
private async void DoComparison(System.IO.Stream imageStream)
{
    Bitmap bmp = null;
    using (StreamImageSource sis = new StreamImageSource(imageStream))
    {
        bmp = await sis.GetBitmapAsync(null, OutputOption.PreserveAspectRatio);

        CustomEffectBase referenceEffect = new CustomEffectManaged.MultiplyEffect(sis,
2);
        DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNative.MultiplyEffect(2));
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNative.MultiplyEffectInlined(2));
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNativeNeonOptimized.MultiplyEffectInlinedOptimized(2));
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNative.MultiplyEffectInlinedUnrolled(2));
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNativeNeonOptimized.MultiplyEffectInlinedUnrolledOptimized(2));
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNativeNeonOptimized.MultiplyEffectInlinedUnrolledOptimizedDeviating(2))
        //DelegatingEffect comparisonEffect = new DelegatingEffect(new
CustomEffectNativeNeonOptimized.MultiplyEffectInlinedUnrolledOptimizedDeviating128(2))
        Comparer.Compare(bmp, referenceEffect, comparisonEffect);
    }
}
```


(C) Copyright Nokia 2014. All rights reserved.

Rendering Deviations

The third page in the user interface shows whether your optimized effect is producing the same output as the reference effect. It also shows any difference in output between the reference effect and the optimized effect. Ideally all values should show 0, meaning that there is no deviation from the reference effect.

The two metrics being used are the maximum deviation encountered during the comparison, and the mean deviation. As you're most often processing the color channels separately, these are given for each of the channels (A = Alpha, R = Red, G = Green, B = Blue), as well as combined for all color components. If you have a high value for Max Deviation and a low value for the Mean Deviation, this means that the comparison effect is producing occasional pixels that are far from the reference rendering, but is doing a fine job for the majority of pixels. A high Mean Deviation means that your effect is overall quite far off from the reference. It is not possible to get a higher Mean Deviation than Max Deviation.

Sometimes you can sacrifice some accuracy to gain a significant speedup in an effect. One thing worth noting though is that the calculated deviations depend on the input image, as well as the parameters you use in your effect. It is entirely possible to get no deviation at all for one input image or a certain set of input parameters, while having a high deviation for another image or set of input parameters. You should therefore test your optimization using multiple images, and/or using images that are designed to make your code enter paths that are likely to lead to a deviation.

If you want to see a practical example of how differences show up in an image, and the resulting deviation numbers, you could remove the processing logic for one of the color components from one of the sample effects:

```
b = MultiplyBounded(b);
//g = MultiplyBounded(g);
r = MultiplyBounded(r);
```

This alteration will exclude the green color component from processing, keeping it the same as in the original image. Therefore only the green channel will show deviations (as well as the overall deviation).

Optimization using SIMD/ARM NEON

ARM NEON is a SIMD instruction set available on all Windows Phone 8 devices. A good introduction to what it is and how it works can be found in this article WP8: Optimizing your signal processing algorithms using NEON. Now I'll explain how to rewrite your code to make it easier to spot ways to use SIMD instructions.

Function Inlining

Function Inlining is an optimization technique commonly used by compilers. Basically, it takes the code of a function/method and copies it in place of the function calls. This can result in performance gains if you have a very short function that is called very often in a loop, such as a function that performs a calculation on each pixel in an image, because it avoids the function call overhead.

The following code inlines our `MultiplyBounded` method to make it easier to see which operations are performed on each color component.

```
27. void MultiplyEffectInlined::Process(Windows::Foundation::Rect rect)
28. {
29.  unsigned int sourceLength, targetLength;
30.  byte* sourcePixels =
    CustomEffectNative::Helpers::GetPointerToPixelData(sourceBuffer, &sourceLength);
31.  byte* targetPixels =
```

 (C) Copyright Nokia 2014. All rights reserved.

```
CustomEffectNative::Helpers::GetPointerToPixelData(targetBuffer, &targetLength);

unsigned int minX = (unsigned int)rect.X * 4;
unsigned int minY = (unsigned int)rect.Y;
unsigned int maxX = minX + (unsigned int)rect.Width * 4;
unsigned int maxY = minY + (unsigned int)rect.Height;

for(unsigned int y = minY; y < maxY; y++)
{
 unsigned int xOffset = y * imageWidth * 4;
 for(unsigned int x = minX; x < maxX; x += 4)
 {
  //Imaging SDK uses Blue, Green, Red, Alpha Image Format with 8 bits/channel
  byte b = sourcePixels[xOffset + x];
  byte g = sourcePixels[xOffset + x + 1];
  byte r = sourcePixels[xOffset + x + 2];
  byte a = sourcePixels[xOffset + x + 3];

```


(C) Copyright Nokia 2014. All rights reserved.

```
    //inlined code from MultiplyBounded

    b = (byte)min(255, ((int)b) * multiplier);

    g = (byte)min(255, ((int)g) * multiplier);

    r = (byte)min(255, ((int)r) * multiplier);

    targetPixels[xOffset + x] = b;

    targetPixels[xOffset + x + 1] = g;

    targetPixels[xOffset + x + 2] = r;

    targetPixels[xOffset + x + 3] = a;

   }

  }

 }
```

In this code, it's easy to see that we're performing exactly the same operations on all three color components. First we multiply each color component. Then we perform a min operation.

Fortunately, the ARM NEON instruction set provides a min operation. We will have to make some changes to our code to take advantage of the ARM NEON instructions.

First, we declare a vector register that holds our multipliers. This vector register is composed of four 16-bit values. The first 3 values are set to the multiplier that we have been using so far. The last value is always set to 1. This is because we will store the alpha value in the fourth position, and we don't want to alter the alpha value.

We declare a similar vector for the min comparison.

Both of these vectors are defined outside the calculation loops, since their values will not change during the computation.

Inside the calculation loops we first store our color components into a vector that holds four 16-bit values. Then, we perform the multiplication and min operations on them, using ARM NEON intrinsics. Finally, we store the resulting data back into the array we initially used to load the data. From there we write the result to the target buffer.

```
void MultiplyEffectInlinedOptimized::Process(Windows::Foundation::Rect rect)
{
 unsigned int sourceLength, targetLength;
 byte* sourcePixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(sourceBuffer,
```

 (C) Copyright Nokia 2014. All rights reserved.

```
&sourceLength);
 byte* targetPixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(targetBuffer,
&targetLength);

 unsigned int minX = (unsigned int)rect.X * 4;
 unsigned int minY = (unsigned int)rect.Y;
 unsigned int maxX = minX + (unsigned int)rect.Width * 4;
 unsigned int maxY = minY + (unsigned int)rect.Height;

 //define an array to hold our multipliers
 uint16_t multArr[4] = { multiplier, multiplier, multiplier, 1 };
 //load the data into a NEON register
 uint16x4_t regMult = vld1_u16(multArr);

 //define an array to hold our min comparison values
 uint16_t minArr[4] = { 255, 255, 255, 255 };
 //load the data into a NEON register
 uint16x4_t regMin = vld1_u16(minArr);

 for(unsigned int y = minY; y < maxY; y++)
 {
  unsigned int xOffset = y * imageWidth * 4;
  for(unsigned int x = minX; x < maxX; x += 4)
  {
   //Imaging SDK uses Blue, Green, Red, Alpha Image Format with 8 bits/channel
   byte b = sourcePixels[xOffset + x];
   byte g = sourcePixels[xOffset + x + 1];
   byte r = sourcePixels[xOffset + x + 2];
   byte a = sourcePixels[xOffset + x + 3];

   //define an array with our pixel data
   uint16_t pixel[4] = { b, g, r, a };
   //load the data into a NEON register
   uint16x4_t regPixel = vld1_u16(pixel);
   //perform multiplication into pixel register
   regPixel = vmul_u16(regPixel, regMult);
   //perform min comparison into pixel register
   regPixel = vmin_u16(regPixel, regMin);
   //restore pixel data to pixel array
   vst1_u16(pixel, regPixel);

   targetPixels[xOffset + x] = (byte)pixel[0];
   targetPixels[xOffset + x + 1] = (byte)pixel[1];
   targetPixels[xOffset + x + 2] = (byte)pixel[2];
   targetPixels[xOffset + x + 3] = (byte)pixel[3];
  }
 }
}
```

We have now performed our first optimization step: moving sequential code to parallel code. Doing another performance test will give us similar results to the following:

Effect	Lumia 1020 max	Lumia 1020 min	Lumia 1520 min
Managed (C#)	745 ms	708 ms	346 ms
Native (C++)	370 ms	358 ms	238 ms

(C) Copyright Nokia 2014. All rights reserved.

Inlined (C++)	274 ms	241 ms	150 ms
Inlined NEON (C++)	381 ms	369 ms	216 ms

Inlining the code has given us a nice speedup compared to the regular C++ version we had before. Using the NEON instructions, however, hasn't given us a performance boost. Instead our effect is performing worse. The reason is that our current NEON implementation requires a copy from memory to memory in order to load the 8-bit color channels as 16-bit values. Given how simple the actual calculation is, the speed gains there don't compensate for the data copying.

We cannot load the data directly from the input buffer and use NEON instructions to do the conversion from 8-bit to 16-bit. The only instructions that allow us to load 8-bit values require eight values, because the vector size has to be 64-bit or 128-bit. We only have 4 values in each loop iteration. Therefore we have to keep looking for other ways to either a) do more parallel computation, or b) find more values so we can load eight of them at once.

Note: We need to use 16-bit values for the multiplication step, because multiplying an 8-bit value by another 8-bit value might lead to an overflow if the target value is also 8-bit wide. However, once the min operation is complete, all values are again normalized to fit inside 8-bit fields.

Loop Unrolling

Loop unrolling means that you merge several iterations of a given loop into one. You can merge an arbitrary number of loop iterations. The number of iterations that you merge is called the unroll factor. You can find a more in-depth description of the unrolling process itself in this article: Wikipedia: Loop unwinding.

regular loop	unrolled loop - factor 2	unrolled loop - factor 3

`//regular loop`
`for (int i = 0; i < 12; i++)`
`{`
`  res[i] = src[i] * multiplier;`
`}`
	`//unrolled loop - factor 2`
	`for(int i = 0; i < 12; i += 2)`
	`{`
	`  res[i] = src[i] * multiplier;`
	`  res[i + 1] = src[i + 1] * multiplier;`
	`}`
		`//unrolled loop - factor 3`
		`for(int i = 0; i < 12; i += 3)`
		`{`
		`  res[i] = src[i] * multiplier;`
		`  res[i + 1] = src[i + 1] * multiplier;`
		`  res[i + 2] = src[i + 2] * multiplier;`
		`}`

If we unroll our inner loop of the inlined version by a factor of 2, we get the following result. Note that we're now incrementing the counter variable x by 8, rather than by 4 as we did in the previous samples.

```
void MultiplyEffectInlinedUnrolled::Process(Windows::Foundation::Rect rect)
{
 unsigned int sourceLength, targetLength;
 byte* sourcePixels = CustomEffectNative::Helpers::GetPointerToPixelData(sourceBuffer, &sourceLength);
 byte* targetPixels = CustomEffectNative::Helpers::GetPointerToPixelData(targetBuffer, &targetLength);
```

 (C) Copyright Nokia 2014. All rights reserved.

```
unsigned int minX = (unsigned int)rect.X * 4;
unsigned int minY = (unsigned int)rect.Y;
unsigned int maxX = minX + (unsigned int)rect.Width * 4;
unsigned int maxY = minY + (unsigned int)rect.Height;

for(unsigned int y = minY; y < maxY; y++)
{
 unsigned int xOffset = y * imageWidth * 4;
 for(unsigned int x = minX; x < maxX; x += 8)
 {
  byte b1 = sourcePixels[xOffset + x];
  byte g1 = sourcePixels[xOffset + x + 1];
  byte r1 = sourcePixels[xOffset + x + 2];
  byte a1 = sourcePixels[xOffset + x + 3];
  //duplicated getter code from unrolling operation
  byte b2 = sourcePixels[xOffset + x + 4];
  byte g2 = sourcePixels[xOffset + x + 5];
```


(C) Copyright Nokia 2014. All rights reserved.

```
    byte r2 = sourcePixels[xOffset + x + 6];

    byte a2 = sourcePixels[xOffset + x + 7];

    b1 = (byte)min(255, ((int)b1) * multiplier);

    g1 = (byte)min(255, ((int)g1) * multiplier);

    r1 = (byte)min(255, ((int)r1) * multiplier);

    //duplicated calculation code from unrolling operation

    b2 = (byte)min(255, ((int)b2) * multiplier);

    g2 = (byte)min(255, ((int)g2) * multiplier);

    r2 = (byte)min(255, ((int)r2) * multiplier);

    targetPixels[xOffset + x] = b1;

    targetPixels[xOffset + x + 1] = g1;

    targetPixels[xOffset + x + 2] = r1;

    targetPixels[xOffset + x + 3] = a1;

    //duplicated setter code from unrolling operation

    targetPixels[xOffset + x + 4] = b2;

    targetPixels[xOffset + x + 5] = g2;
```

 (C) Copyright Nokia 2014. All rights reserved.

```
    targetPixels[xOffset + x + 6] = r2;
```

```
    targetPixels[xOffset + x + 7] = a2;
```

```
   }
```

```
  }
```

```
 }
```

Our unrolled loop is now processing two pixels at a time, which means it's processing eight 8-bit color components. This means that we can load a whole 64-bit vector from the source buffer using a NEON load operation, avoiding the memory-to-memory copy we had to resort to in our previous optimization attempt.

The two registers that contain the operands for the multiplication consequently consist of eight 16-bit values, rather than four. Inside the calculation loop, we perform a `movl` operation to convert the 64-bit vector that contains the eight 8-bit input values into a 128-bit vector that contains eight 16-bit values. The 128-bit vector is first multiplied, then its separate values are bounded to 255 using a min operation. At this point we can use a casting operation to convert it back to a 64-bit vector.

Lastly, we write that vector directly to the output buffer.

```
void MultiplyEffectInlinedUnrolledOptimized::Process(Windows::Foundation::Rect rect)
{
 unsigned int sourceLength, targetLength;
 byte* sourcePixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(sourceBuffer,
&sourceLength);
 byte* targetPixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(targetBuffer,
&targetLength);

 unsigned int minX = (unsigned int)rect.X * 4;
 unsigned int minY = (unsigned int)rect.Y;
 unsigned int maxX = minX + (unsigned int)rect.Width * 4;
 unsigned int maxY = minY + (unsigned int)rect.Height;

 //define an array to hold our multipliers
 uint16_t multArr[8] = { multiplier, multiplier, multiplier, 1, multiplier, multiplier,
multiplier, 1 };
 //load the data into a NEON register
 uint16x8_t regMult = vld1q_u16(multArr);

 //define an array to hold our min comparison values
 uint16_t minArr[8] = { 255, 255, 255, 255, 255, 255, 255, 255 };
 //load the data into a NEON register
 uint16x8_t regMin = vld1q_u16(minArr);

 for(unsigned int y = minY; y < maxY; y++)
 {
  unsigned int xOffset = y * imageWidth * 4;
  for(unsigned int x = minX; x < maxX; x += 8)
  {
```

 (C) Copyright Nokia 2014. All rights reserved.

```
    //load pixel data of two adjacent pixels into NEON register
    uint8_t* pixel_8 = (uint8_t*)&sourcePixels[xOffset + x];
    uint8x8_t regPixel_8 = vld1_u8(pixel_8);
    //convert 8 8-bit registers to 8 16-bit registers
    uint16x8_t regPixel_16 = vmovl_u8(regPixel_8);
    //perform multiplication into pixel register
    regPixel_16 = vmulq_u16(regPixel_16, regMult);
    //perform min comparison into pixel register
    regPixel_16 = vminq_u16(regPixel_16, regMin);
    //convert 8 16-bit registers to 8 8-bit registers
    regPixel_8 = vqmovn_u16(regPixel_16);
    //store pixel data of two adjacent pixels into targetPixels
    pixel_8 = (uint8_t*)&targetPixels[xOffset + x];
    vst1_u8(pixel_8, regPixel_8);
   }
  }
 }
```

Once we've resolved the memory copying issue we encountered during the last optimization, we can take another look at our performance metrics:

Effect	Lumia 1020 max	Lumia 1020 min	Lumia 1520 min
Managed (C#)	745 ms	708 ms	346 ms
Native (C++)	370 ms	358 ms	238 ms
Inlined (C++)	274 ms	241 ms	150 ms
Inlined NEON (C++)	381 ms	369 ms	216 ms
Inlined Unrolled (C++)	240 ms	226 ms	146 ms
Inlined Unrolled NEON (C++)	203 ms	193 ms	99 ms

Looking at these numbers, it's clear that unrolling our sequential code by a factor of two results in a slight performance boost. In addition, by using NEON instructions, we were able to not only beat the previous NEON version, but also to finally take advantage of the parallel computation to beat the sequential version.

Note: By unrolling the loop by a factor of two, we assume that the width of an input image is an even number. For odd widths we would always read/write one pixel too many for each row. To keep the sample code simple, this case is not handled. In my experience you will rarely see images that have an odd width. If you think your code might have to deal with this case, you should check for it and at least inform the user of the issue, or better yet add another single loop iteration that will be executed for the last pixel in each row.

Optimizations leading to deviations in the result

So far all our optimizations were able to produce exactly the same result as our reference effect. Sometimes, you can speed up your calculations quite a bit by using tricks that will result in a slightly different result. While that would not be acceptable in many use cases, such as financial data, it will often not be noticeable in image processing.

One issue with our implementation so far is that we have to convert our 8-bit values to 16-bit values in order to handle a possible overflow scenario. Alternately, we could manipulate the input values of the multiplication in such a way that an overflow is no longer possible. This can be done by setting any value that would lead to an overflow to the biggest value that does not produce the overflow.

If we use the multiplier 2, then we set every value in the input buffer that is bigger than 127 to 127. In general, this means that we set any value in the input buffer that is bigger than 255 / multiplier to 255 / multiplier. Effectively, we're now doing the min operation before the multiplication.

The result of any number multiplied by 2 following the original min operation would have been 255. Using our optimization, however, the result is now 254. This means we no longer need to move our data between 64bit and 128bit vectors, thus simplifying our code.

 (C) Copyright Nokia 2014. All rights reserved.

```
void MultiplyEffectInlinedUnrolledOptimizedDeviating::Process(Windows::Foundation::Rect
rect)
{
 unsigned int sourceLength, targetLength;
 byte* sourcePixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(sourceBuffer,
&sourceLength);
 byte* targetPixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(targetBuffer,
&targetLength);

 unsigned int minX = (unsigned int)rect.X * 4;
 unsigned int minY = (unsigned int)rect.Y;
 unsigned int maxX = minX + (unsigned int)rect.Width * 4;
 unsigned int maxY = minY + (unsigned int)rect.Height;

 //define an array to hold our multipliers
 uint8_t multArr[8] = { multiplier, multiplier, multiplier, 1, multiplier, multiplier,
multiplier, 1 };
 //load the data into a NEON register
 uint8x8_t regMult = vld1_u8(multArr);

 byte bound = 255 / multiplier;
 //define an array to hold our min comparison values
 uint8_t minArr[8] = { bound, bound, bound, 255, bound, bound, bound, 255 };
 //load the data into a NEON register
 uint8x8_t regMin = vld1_u8(minArr);

 for(unsigned int y = minY; y < maxY; y++)
 {
  unsigned int xOffset = y * imageWidth * 4;
  for(unsigned int x = minX; x < maxX; x += 8)
  {
   //load pixel data of two adjacent pixels into NEON register
   uint8_t* pixel = (uint8_t*)&sourcePixels[xOffset + x];
   uint8x8_t regPixel = vld1_u8(pixel);
   //perform min comparison into pixel register
   regPixel = vmin_u8(regPixel, regMin);
   //perform multiplication into pixel register
   regPixel = vmul_u8(regPixel, regMult);
   //store pixel data of two adjacent pixels into targetPixels
   pixel = (uint8_t*)&targetPixels[xOffset + x];
   vst1_u8(pixel, regPixel);
  }
 }
}
```

We are now fully utilizing our 64-bit vectors during the computation. However, we have 128-bit vectors available, and we have enough data to fill them. If we unroll our loop by a factor of 4, rather than a factor of 2, we can load 16 8-bit values at once, thus processing four pixels at a time. The code for doing so is shown below.

```
void
MultiplyEffectInlinedUnrolledOptimizedDeviating128::Process(Windows::Foundation::Rect
rect)
{
```

 (C) Copyright Nokia 2014. All rights reserved.

```
 unsigned int sourceLength, targetLength;
 byte* sourcePixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(sourceBuffer,
&sourceLength);
 byte* targetPixels =
CustomEffectNativeNeonOptimized::Helpers::GetPointerToPixelData(targetBuffer,
&targetLength);

 unsigned int minX = (unsigned int)rect.X * 4;
 unsigned int minY = (unsigned int)rect.Y;
 unsigned int maxX = minX + (unsigned int)rect.Width * 4;
 unsigned int maxY = minY + (unsigned int)rect.Height;

 //define an array to hold our multipliers
 uint8_t multArr[16] = { multiplier, multiplier, multiplier, 1, multiplier, multiplier,
multiplier, 1, multiplier, multiplier, multiplier, 1, multiplier, multiplier,
multiplier, 1 };
 //load the data into a NEON register
 uint8x16_t regMult = vld1q_u8(multArr);

 byte bound = 255 / multiplier;
 //define an array to hold our min comparison values
 uint8_t minArr[16] = { bound, bound, bound, 255, bound, bound, bound, 255, bound,
bound, bound, 255, bound, bound, bound, 255 };
 //load the data into a NEON register
 uint8x16_t regMin = vld1q_u8(minArr);

 for(unsigned int y = minY; y < maxY; y++)
 {
  unsigned int xOffset = y * imageWidth * 4;
  for(unsigned int x = minX; x < maxX; x += 16)
  {
   //load pixel data of two adjacent pixels into NEON register
   uint8_t* pixel = (uint8_t*)&sourcePixels[xOffset + x];
   uint8x16_t regPixel = vld1q_u8(pixel);
   //perform min comparison into pixel register
   regPixel = vminq_u8(regPixel, regMin);
   //perform multiplication into pixel register
   regPixel = vmulq_u8(regPixel, regMult);
   //store pixel data of two adjacent pixels into targetPixels
   pixel = (uint8_t*)&targetPixels[xOffset + x];
   vst1q_u8(pixel, regPixel);
  }
 }
}
```

In addition to moving to the 128-bit vector operations, we again have to change our increment of x; this time to 16 instead of 8. It is worthwhile to always check this when doing loop unrolling. (I forgot to do so myself while writing the sample.)

It is now time for the last performance comparison table on our own optimizations to see what this deviating performance optimization has achieved.

Effect	Lumia 1020 max	Lumia 1020 min	Lumia 1520 min
Managed (C#)	745 ms	708 ms	346 ms
Native (C++)	370 ms	358 ms	238 ms
Inlined (C++)	274 ms	241 ms	150 ms

(C) Copyright Nokia 2014. All rights reserved.

Inlined NEON (C++)	381 ms	369 ms	216 ms
Inlined Unrolled (C++)	240 ms	226 ms	146 ms
Inlined Unrolled NEON (C++)	203 ms	193 ms	99 ms
Inlined Unrolled Deviating NEON 64 bit (C++)	184 ms	164 ms	70 ms
Inlined Unrolled Deviating NEON 128 bit (C++)	125 ms	111 ms	51 ms

As you can see, by sacrificing a little accuracy, our last NEON implementation is now around twice as fast as the pure C++ implementation. Given how small the deviations from the reference rendering are, the gains in performance might very well be worth the tradeoff. (The deviations page is showing a maximum deviation of 1 with a mean deviation far smaller while using the multiplication factor of 2). More information on optimizations that create slight variation from the reference result while computing quite a bit faster can be found in the optimization section of this article: Image processing optimization techniques

Compiler Optimizations

As I already mentioned when discussing function inlining, many of the techniques we use to identify potential for parallelizing code are actually employed by optimizing compilers themselves to make the code as fast as possible. So why do we see performance differences between our non-inlined and inlined C++ code? The answer is easy - I have done all comparisons so far using debug code, which is not only slower overall due to the debugging symbols, but also effectively keeps the compiler from doing optimizations itself. The reason to do this was to showcase what effect those changes can have on their own, given that for more complex code the compiler itself might not be able to do some optimizations, such as loop unrolling (especially on nested loops), which a programmer can still do himself.

So let's have a look at how our samples perform when compiled in the release configuration.

Effect	Lumia 1020 debug	Lumia 1020 release	Lumia 1520 release
Managed (C#)	708 ms	572 ms	244 ms
Native (C++)	358 ms	81 ms	45 ms
Inlined (C++)	241 ms	81 ms	45 ms
Inlined NEON (C++)	369 ms	94 ms	56 ms
Inlined Unrolled (C++)	226 ms	78 ms	45 ms
Inlined Unrolled NEON (C++)	193 ms	66 ms	38 ms
Inlined Unrolled Deviating NEON 64 bit (C++)	164 ms	62 ms	36 ms
Inlined Unrolled Deviating NEON 128 bit (C++)	111 ms	62 ms	36 ms

Aside from the fact that the release code, especially the native code, is a lot faster, you can immediately see that some of our optimization steps that are not moving code from C# to C++, or from pure C++ to C++/NEON, are not having much of an effect. This is due to the compiler optimizations. You can also see that the compiler is not able to protect us from issues like the additional memory copy we did during our first attempt at utilizing the NEON instructions. The last thing you might notice is that percentage-wise, the gains seem to be much smaller in the release configuration than they were in the debug configuration. This can be explained if you keep in mind that we're not measuring our effect's execution itself, but the time for the whole rendering pipeline surrounding it. (Most of the time is spent copying data into our source buffer and out of our target buffer). The time it takes for the SDK to perform those tasks isn't affected by our change in configuration, since we're always linking to the SDK release code. This time, however, is now making up a bigger percentage of the time the rendering pipeline takes to execute.

When to optimize

Having read all this you might now ask yourself: when do I start optimizing my code? When do I stop? In my opinion it is best to start implementing your custom effect in a straightforward, easy-to-understand way. Then you should test if it is fast enough. If it's already fast enough, I don't see much reason to start putting a lot of effort into optimization.

However, as you have seen, it is quite easy to move your code from C# to C++, and you get quite a big boost in performance by doing so. So this is something you should always consider doing.

Moving to SIMD instructions or applying other (potentially lossy) optimizations, however, is something I suggest you don't do, unless your use case requires it (for example, for realtime camera feed preview), due to the fact that is easy to spend a considerable amount of time on optimization.

 (C) Copyright Nokia 2014. All rights reserved.

Summary

Starting out with a very simple effect implemented in C# that took around 700 ms to complete, we arrived at a solution that takes consistently less than a sixth of the time to complete using C++ native code and ARM NEON intrinsics. On the way we showed how to implement effects for the Nokia Imaging SDK using C++, and how to compare the performance and image fidelity of different versions of a filter. We also showed two ways to rewrite code that make it easier to spot opportunities to make use of ARM NEON instructions. On the way we encountered a situation where, in order to use ARM NEON instructions, we had to do conversions that were expensive enough to make the actual performance of the effect worse. Continuing on that path, however, we were able to build the fastest version of our effect that gives exactly the same result as the initial effect. We then implemented a change that, while leading to a small change in the output, allowed us to almost double the performance again, compared to the already fast previous version.

Optimization experts might be able to improve performance even more.

The performance improvements of our ARM NEON-optimized code over the pure C++ code are quite nice. However, due to the simple nature of our effect, which requires only a few calculations while doing a lot of loads/stores from/to memory, the gains in this application don't begin to approach the performance gains you can achieve on the more complex effects that often come up in discussions on the possible performance gains using SIMD instructions.

In the end we discussed how compiler optimization plays into this, and why it is still worthwhile (and often necessary) to do optimizations yourself.

Final words

If you run into issues with the samples, find a bug or know of a technique that could be applied towards optimizing our `MultiplyEffect` please add a comment or extend the article accordingly.

 (C) Copyright Nokia 2014. All rights reserved.